Helix: IP Lookup Scheme based on Helicoidal Properties of Binary Trees

Roberto Rojas-Cessa[a,*], Taweesak Kijkanjanarat[b], Wara Wangchai[c], Krutika Patil[a], Narathip Thirapittayatakul[b]

[a]*Department of Electrical and Computer Engineering, New Jersey Institute of Technology, Newark, NJ, 07102, U.S.A.*
[b]*Department of Electrical and Computer Engineering, Thammasat University, Rangsit, Pathumtani, Thailand.*
[c]*Department of Computer Engineering, Department of Engineering, Kasetsart University, Thailand.*

Abstract

In this paper, we propose an IP lookup scheme, called Helix, that performs parallel prefix matching at the different prefix lengths and uses the helicoidal properties of binary trees to reduce tree height. The reduction of the tree height is achieved without performing any prefix modification. Helix minimizes the amount of memory used to store long and numerous prefixes and achieves IP lookup and route updates in a single memory access. We evaluated the performance of Helix in terms of the number of memory accesses and amount of memory required for storing large IPv4 and IPv6 routing tables with up to 512,104 IPv4 and 389,956 IPv6 prefixes, respectively. In all the tested routing tables, Helix performs lookup in a single memory access while using very small memory amounts. We also show that Helix can be implemented on a single field-programmable gate array (FPGA) chip with on-chip memory for the IPv4 and IPv6 tables considered herein, without requiring external memory. Specifically, Helix uses up to 72% of the resources of an FPGA to accommodate the most demanding routing table, without performance penalties. The implementation shows that Helix may achieve lookup speeds beyond 1.2 billion packets per second.

Keywords:
IP lookup, binary tree, one memory access, helicoidal properties, double helix, IPv6 lookup.

1. Introduction

Internet Protocol (IP) address lookup is one of the different tasks that an Internet router performs on traversing packets [1]. It is the process of finding the output port where to forward a packet from a forwarding table. Because the table is implemented on speed-lagging memory, IP lookup must be efficiently performed to avoid making IP lookup a performance bottleneck [2].

Several factors make IP lookup a challenging task: (a) The increasing transmission speed of links, currently at 100 Gbps, demands packet processing running at similar high speeds [3]. (b) Routing tables keep growing at a very fast pace. For example, the Border Gateway Protocol (BGP) table for autonomous system 6447 (AS6447) recorded on May 8, 2014 holds 512,104 IPv4 prefixes and 17,223 IPv6 prefixes [4]. Prefix matching must keep up with the increasingly larger number of prefixes. (c) The address length of the forthcoming IPv6 is 128 bits. Therefore, scalability of IP lookup schemes toward IPv6 cannot be longer neglected. Many of the existing IP lookup schemes were designed for implementation on hardware or software while IPv4 was being rolled out. Scalability for IPv6 prefixes may not have been considered then but for recent schemes [5, 6, 7, 8, 9].

Because of the slow speed of memory, minimizing the number of memory accesses required to perform IP lookup is a major performance goal. The size of the data structure, or amount of memory, representing a forwarding table for IP lookup is another important parameter. To run at the fastest achievable speed, the required amount of memory must be small enough to fit in on-chip memory [10]. Furthermore, routing tables undergo frequent updates. It has been reported that backbone routers may experience up to hundred thousand updates per second [2, 11, 12, 13, 14, 10].

[*]Corresponding author.
Email addresses: `rojas@njit.edu` (Roberto Rojas-Cessa), `taweesak@engr.tu.ac.th` (Taweesak Kijkanjanarat), `warawangchai@gmail.com` (Wara Wangchai)

The time required to update routing tables may hamper the highest achievable performance for IP lookup as changes are accommodated at run time. Therefore, increasing lookup and update speed, and reducing memory size are the objectives for achieving high performance in IP lookup engines [15, 16].

Research on IP lookup schemes has been extensive in recent years. The design objectives of a large number of the existing schemes target fast IP lookup, memory minimization, or a combination of both [17, 18, 2, 19, 20, 21, 22, 23, 24]. These schemes are mainly based on representing prefixes in binary trees and on derivatives of them. However, as prefix length increases, so does the number of memory accesses required to perform the longest prefix matching.

Therefore, there is a need of schemes able to perform IP lookup and table updates in the ideal time of one memory access while using a small amount of memory for routing tables with a large number of IPv4 and IPv6 prefixes.

In this paper, we address this need by proposing an IP lookup scheme, named Helix. The proposed scheme performs both IP lookup and table updates for IPv4 and IPv6 prefixes in a single memory access. This scheme is the only one that has reported this optimum lookup speed, to the best of our knowledge. Helix achieves this lookup speed using very small amounts of memory. Helix uses parallel prefix search [25, 26, 27] at the different prefix lengths and the helicoidal properties of binary trees, introduced herein. Helix is tested under a legacy and recent IPv4 routing tables, and a synthetic IPv6 routing table. The legacy IPv4 routing table, recorded on August 24, 1997, holds 39K prefixes. The recent IPv4 routing table, recorded on May 8, 2014 holds 512K prefixes and the IPv6 routing table, which is derived from the IPv4 table of AS65000, recorded also on May 8, 2014, holds 389K prefixes. In all tested cases, Helix performs lookup in a single memory access and resorts to a very small amount of memory to store the prefixes. We show that the memory required by Helix for the tested routing tables fits in a single field-programmable gate array (FPGA) chip, while keeping the lookup and update speeds at one memory access. We also show that the efficiency of Helix supports IP lookup for very high data rates and requires very small amount of gate resources of an FPGA chip.

The remainder of this paper is organized as follows. Section 2 presents related works. Section 3 introduces the proposed IP lookup scheme and the helicoidal properties of binary trees. Section 4 presents an evaluation of the performance of Helix under a legacy and recent IPv4 routing tables, and an IPv6 routing table in terms of lookup and table-update speeds. This section also present the amount of memory used for the tested routing tables. Section 5 discusses the implementation of Helix on a single FPGA for the studied IPv4 and IPv6 routing tables. Section 6 presents a brief comparison of the memory usage of Helix and other memory-efficient schemes. Section 7 presents our conclusions.

2. Related Work

IP lookup has been of research interest for several years, and although the number of contributions in the literature is rather large, herein we focus on those schemes whose working principles are related to the proposed scheme. Readers are referred to surveys for exhaustive listings of IP lookup schemes [28, 29, 30].

We broadly categorize IP lookup schemes into ternary content addressable memory (TCAM) and binary-tree based approaches. TCAM-based schemes ably achieve the optimum number of memory accesses, or one, but they may accommodate a limited number of prefixes, consume large amounts of power, and require large access times. Schemes aimed at reducing TCAM power consumption have been developed but they may also penalize the number of prefixes that can be stored or the access time [31].

Binary-tree schemes don't share the same hindrances of TCAM-based schemes but they may take a larger number of memory accesses to perform lookup. In a straight-forward application of a binary tree for IP lookup, each bit of a packet destination is compared to each bit of a binary-tree on a leaf-ward direction. This scheme takes as many memory access times as the number of bits of the longest prefix in the table. The Path-compressed (PATRICIA) trie scheme reduces the number of accesses by comparing multiple bits at once [32]. The Level-Compressed (LC) trie may achieve further compression of nodes towards a prefix node by reducing the number of levels of the tree [33]. In fact, the LC-trie is a scheme that has prevailed as a benchmark for a long time because of its achievable compression gain. The compression of levels is traded for an increase in the number of children nodes, converting a binary tree into a γ-ary tree. A level can be compressed if there are no intermediate prefixes between the target levels. However, high prefix density may hinder the compression of a large number of bits as nodes branch out, affecting the efficacy of PATRICIA and LC tries. Nevertheless, these two approaches show that the compression of bits of a binary tree can be pursued in a vertical (path compression) or horizontal (level compression) fashion.

An interval-based technique, where prefixes are sorted and used to determine the boundaries of numeric intervals, may facilitate comparisons of multiple prefixes at once [34]. In this technique, packet destinations are also represented as binary numbers and placed in the corresponding interval. If the packet destination is different from the binary number represented by the node, then the packet destination belongs to the interval of prefixes whose numerical value is larger (or smaller) than the destination. This technique reduces the number of comparisons, the tree height, and the number of memory accesses, but the intervals are defined by the prefixes in the table. Furthermore, route updates are complex to compute. This high complexity dampers the scalability and the lookup speed of the scheme.

The combination of path compression with bit-map representation, where a sub-tree is considered as a compound node, may reduce a tree's height [35, 13]. Bit maps are used to indicate the value of the prefix by the bit position of a prefix (i.e., prefixes are represented as numbers in an interval) on a tree level [36, 37]. This approach reduces the amount of memory used to represent the prefixes of a given length. However, this approach may not be fast enough to keep up with the increasing link speeds, as fetching long prefixes may still require multiple memory accesses. Route updates are also complex to perform in bitmap-based schemes as schemes resort to prefix modification (e.g., leaf pushing).

Representing prefixes at a selected level is an alternative to path compression. Using a limited number of tree levels may reduce the amount of memory used to represent all the prefixes [36], but it may not reduce access time effectively. In the reduction of the number of the tree levels, prefixes in smaller levels are expanded to a larger selected level. The representation of the prefixes in the largest tree level of a forwarding table remains challenging, as it requires very tall trees. The height of the tree is determined by the distance between the tree's root to the farthest leaf. Therefore, keeping the original levels of the binary tree avoids congesting a level with replicated prefixes and makes it easy to perform table updates [26].

Hashing prefixes may reduce the representation of them (by selecting a number of bits that can be used to address the memory location of the prefix), where the common bits of the prefixes are selected as a hash index for matching [38, 39]. The selection of a hash index based on the actual prefixes may be more efficient than using an arbitrary function (e.g., a cyclic-redundancy check polynomial) [39]. In hashing, prefix expansion may be undesirable for current forwarding tables as the number of prefixes is large and this increases the number of collisions (i.e., the number of prefixes that are indexed by the same hash index). Bloom filters can be used to increase hashing efficiency by eliminating false negatives [40, 41, 42, 43]. However, they may not eliminate false positives.

Recent schemes combine several techniques to reduce memory footprint [44, 45, 10]. Prefix partitioning has been recently proposed, where partial prefix expansion and range search are combined [44]. However, the lookup speed of this scheme is similar to that of range search schemes but memory usage is not optimum. Shape graphs is another approach to reduce the amount of the memory used to represent a routing table [45]. The shape-graph approach uses the distribution of nodes of a binary tree to reduce the number of nodes to be represented. However, performing updates is complex in this approach. Furthermore, the computation complexity of the analysis of prefix distribution in large tables may be high. An approach where hashing is combined with node compression has been recently proposed [10]. This scheme stores Next Hop IDs (NHIs) according to the distribution of the routing table to reduce the memory amount used to represent intermediate trie nodes. The Offset Encoded Trie (OET) is a recent scheme that minimizes the representation of a Multibit trie [46] and, therefore, reduces the needed memory amount [47]. OET requires complex processing of the binary tree to minimize the amount of memory used in the representation of prefixes. This scheme pursues memory reduction rather than lookup speed as it is aimed for use on virtualized software routers. These recent schemes, nevertheless, require multiple memory accesses to perform lookup in the worst case scenario.

In our proposed methodology, we use the helicoidal properties of binary trees to reduce tree height or memory amount. In the search process, we perform lookup at each reduced level, therefore, holding prefixes intact for location ease. This representation of prefixes is memory efficient and enables quick access to prefixes.

3. Proposed Scheme and Terminology

3.1. IP Lookup

In a forwarding table, each prefix is associated with a NHI, which is the port where a packet is forwarded after matching a given prefix. A forwarding table has as many different NHIs as the number of the ports, k, of the router hosting the table. Table 1 shows an example of a forwarding table, which is used as an example throughout this paper

IP prefix	prefix ID
0*	a
000*	b
111*	c
0000*	d
00001*	e
000001*	f
000110*	g
1110*	h
1111*	i
111001*	j
11101*	k
111111*	l

Table 1: Prefixes of an example forwarding table.

to describe the proposed scheme and the helicoidal properties. Herein, we use the prefix identification label instead of the NHI for description simplicity. Figure 1 shows the prefixes listed in Table 1 as a binary tree. The height of this tree is six, and the tree is also referred to as having six levels. Therefore, prefix length y of prefix x is referred to as the level of the tree where the prefix is located. In this example, prefix **a** (x =0*) is located at level 1 (y = 1), and prefix **l** (x =111111*) is located at level 6 (y = 6).

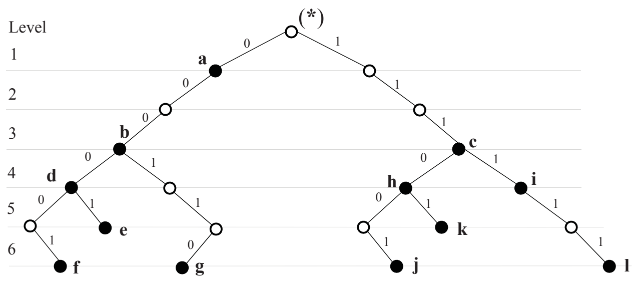


Figure 1: Binary tree of the prefixes in Table 1.

3.2. IP Lookup Process

We first identify the prefix (or tree) levels that have one or more prefixes, or non-empty prefix levels. Prefixes are kept in their original levels; prefix lengths remain unchanged. We allocate a prefix table for each non-empty prefix level, where each table is implemented as a memory block. Note that keeping the prefixes with their original length has the following benefits: 1) the number of prefixes in the binary tree does not increase and 2) it is simple to make table updates.

Prefixes of a large level may include a large number of prefixes; 2^y, and they must be efficiently stored in the corresponding table. We achieve this objective by using the helicoidal properties of binary trees, introduced in this section.

Lookup process: After a packet arrives, the destination address (D) of the packet is used to search a matching prefix in all different prefix tables, simultaneously. This approach is called *parallel search matching* [26]. Among the matched prefixes at the different levels, the longest prefix is selected.

Figure 2 shows the top-level IP lookup process. This example considers IPv4 prefixes. In this figure, there is a prefix table for each non-empty prefix level. A matched prefix, if any, is output by the corresponding level table(s), and the selector selects the longest matching prefix.

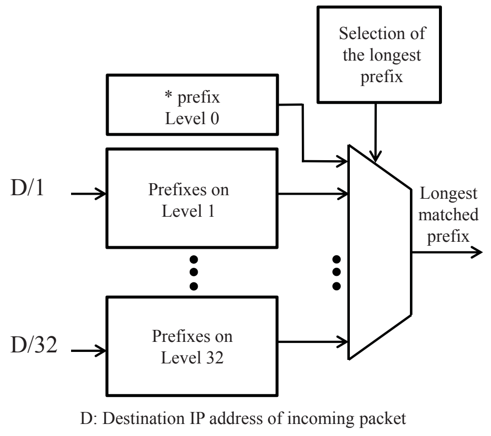


Figure 2: Parallel-level lookup process.

This process performs exact prefix matching at each prefix level. For that, we aim to use a prefix, or a portion of it, as an address of an entry of a prefix table.

In a prefix table, each location stores the NHI of the prefix assigned to that location. In general, the memory size of the prefix table for level v, ($S(v)$), is

$$S(v) = 2^v \log_2 k \tag{1}$$

where 2^v is the number of locations in the table. This number is referred to as the *memory length*, and the number of bits stored in the location is referred to as the *memory width*. In this case, the memory width is $\log_2 k$, which is the number of bits representing an NHI. The amount of memory used by a prefix table is the memory length times the memory width. Note that this table provisions space for the largest number of prefixes at the level rather than the amount of memory used by the exact number of prefixes.

The binary tree of Figure 1 has six levels but only five levels hold prefixes; levels 1, 3, 4, 5, and 6. Therefore, five prefix tables are provisioned. The table for level 6 requires 64 memory locations and holds prefixes **f**, **g**, **j**, and **l**. In general, IPv4 and IPv6 forwarding tables, where $0 \leq y \leq 32$ and $0 \leq y \leq 128$, respectively, may need very long memories for the largest levels. Moreover, fast memory may be costly, or even infeasible, for those levels. For example, a prefix table for level 128 for IPv6 would require 2^{128} locations.

As (1) shows, the memory length increases exponentially as the level increases. Therefore, the memory length is the dominant factor in the determination of the amount of used memory [22, 10]. On the other hand, the memory width is a factor that may affect the implementation with discrete devices and memory (access) speed [48]. The helicoidal properties, introduced in the remaining of this section, are used for representing long prefixes in small prefix tables (and small memories).

3.3. Representation of Prefixe in a Helix Data Structure

Because prefixes are keep in the form in which they are reported in routing tables, a prefix may be directly used as address to a location in a prefix table. Helix minimizes the number of bits used to address the location of prefix x in a prefix table by splitting the prefix into two parts: the Most Significant Bits (MSBs) portion or *Family root (FRx)* of x and the Least Significant Bits (LSBs) portion or *Descendant bits (Dx)* of x. Figure 3 shows these two portions of bits in a prefix, separated between bits of level T and $T + 1$. FRx is stored with the corresponding NHI in the location addressed by Dx.

The line between FRx and Dx in the figure, called the torsion level (L_T), indicates the bit where the original prefix is split. The selection of L_T determines the size of FRx and Dx, in number of bits, and subsequently the size of the memory amount used for prefix level v. We also describe these terms by the role they play on a binary tree.

As an example, let's consider level 6 of the example binary tree after applying a torsion on level 3 ($L_T = 3$). In this case prefixes on level 6, **f**, **g**, **j**, and **l**, are represented as $FRf = 000$, $FRg = 000$, $FRj = 111$, and $FRl = 111$, while $Df = 001$, $Dg = 110$, $Dj = 001$, and $Dl = 111$. Since Dx is used to address the location of prefixes, we see

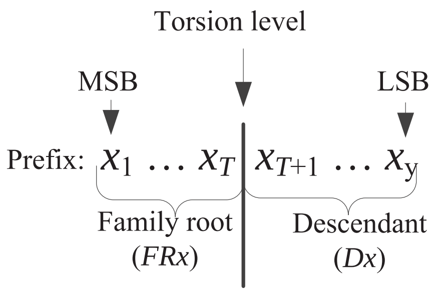
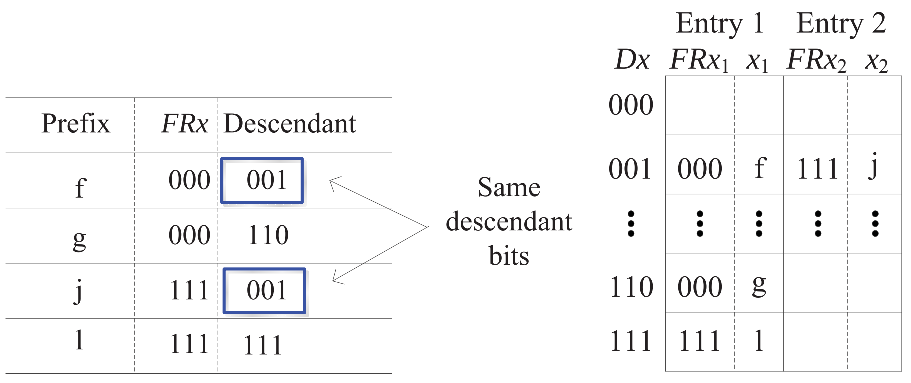


Figure 3: Parts of a prefix for Helix representation.

that multiple prefixes may share the memory location. In our example, prefixes **f** and **j** share the same location. These prefixes are said to be *conjoined*. Figures 4(a) and 4(b) show a prefix table with **f** and **g** sharing the same location.


Same descendant bits

Prefix	FRx	Descendant
f	000	001
g	000	110
j	111	001
l	111	111

(a)

	Entry 1		Entry 2	
Dx	FRx_1	x_1	FRx_2	x_2
000				
001	000	f	111	j
⋮	⋮	⋮	⋮	⋮
110	000	g		
111	111	l		

(b)

Figure 4: (a) FRx and Dx parts of prefixes at level 6 after $L_T = 3$. (b) Reduced prefix table of level 6.

To show a perspective of a torsion and the subtrees it forms in a binary tree, we add some non-existing nodes to the example binary tree, as Figure 5(a) shows. These nodes are shown with a dashed-line circumference. A torsion level on a binary tree can be described as the level of tree where all links towards the descendants of the nodes at L_T are spined by an angle α in a counter-clock-wise direction. The actual value of α is irrelevant for this discussion, but for the sake of description, it may be considered as 90 degrees in a counter-clockwise direction, as Figure 5(b) shows.

This figure also shows the torsion on level 3 of the example the binary tree, the resulting subtrees below the torsion level, and the representation of conjoined prefixes on level 6 of the resulting tree. The nodes below L_T form subtrees rooted to their string of ancestor nodes; from the root (*) on level 0 to L_T. This string of bits is FRx. The remaining string of bits is Dx.

The left side of this figure shows the subtrees generated by the torsion, which produces two subtrees rooted at prefixes **b** and **c** as they are the only nodes at L_T with descendant prefixes. The appearance of the binary tree resembles a double helix, as there are two subtrees, one rooted to **b** and the other rooted to **c**. Each one of the two subtrees (or helices) has a different family root.

Subtrees are superposed to represent all subtrees after a torsion as one tree. Here, superposition is the process of placing the nodes of coplanar subtrees on the same position. The subtrees rooted at L_T in Figure 5(b) are superposed, or represented as a single tree. The right side of Figure 4(b) shows the prefix table for level 6 obtained after superposition. From superposition, two or more conjoined prefixes may occupy the same position on a prefix level.

By adding conjoined prefixes, the memory length is reduced and the memory width is increased. As the memory length increases, the total amount of memory exponentially increases. Therefore, keeping the memory length small helps to save large amounts of memory. The remainder binary trees show conjoined prefixes as filled grey color

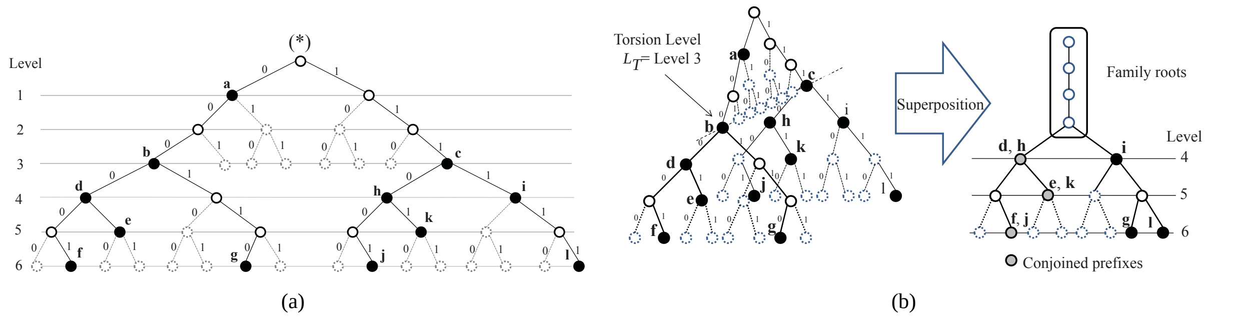


Figure 5: (a) Example binary tree fitted with additional nodes. The dashed-line nodes are added for aiding the visualization of subtrees. (b) Levels 4, 5, and 6 form two subtrees after a torsion, which are superposed.

circles. On the other hand, a prefix that does not share its location with any other node is a non-conjoined prefix. A non-conjoined prefix is shown as a black node in the binary trees discussed in the remainder of this paper. On level 6, prefixes **g** and **l** are non-conjoined prefixes and prefixes **f** and **j** are conjoined prefixes, as this figure shows.

Note that the resulting tree in this figure is shorter than the original tree. For example, the height of the original tree (Figure 1) is six, while the height of the tree in right side of Figure 5(b) is three (the compound node FRx counts as a single node).

3.3.1. Contents of a Location in a Prefi Table

The resulting data structure for representing a prefix level after a torsion is similar to that used to represent the original prefixes. The data structure of Helix takes contiguous MSB bits selected after a torsion and store them in the table entry for each prefix. Therefore, the complexity of the data structure is $O(1)$ as no operations need to be performed on the stored prefixes, and Dx directly addresses x in a prefix table. Many other schemes resort to prefix expansion, where prefixes have to be replicated, up to 2^h times, where h is the difference in bits between the expanded level and the original prefix length, increasing the complexity then to $O(2^h)$ [34, 33, 10].

A prefix table allocates one location per node position at level v. The width and contents of the prefix table for level v depend on the selected L_T. The number of bits of a location is defined by the NHI bits, the largest number of conjoined prefixes (q), and the number of bits of family roots (i.e., L_T bits).In summary, there are two data formats, one for $L_T = 0$ and the other level for $L_T > 0$:
1) $L_T = 0$**.** This is the case when the representation of a prefix level does no resorts to the use of a torsion; the shape of the tree remains as in the original tree. In this case, there are only non-conjoined prefixes in the prefix table. Figure 6(a) shows the contents of a location of the prefix table. Each location of the table for any of these levels stores the NHI of the corresponding prefix and x is used to address a location in the table. The size of the prefix table is determined by (1).
2) $L_T > 0$**.** In this case, the prefix table may include both non-conjoined and conjoined prefixes. Figure 6(b) shows the content format of a table location. Here, Dx is the location address. The NHI is stored with the FRx for each prefix, with the latter being used to verify the match. The size of a prefix table with $q > 0$ is

$$S(v) = 2^{(v-L_T)}q(L_T + \log_2 k), \tag{2}$$

where L_T also indicates the number of bits of FRx. If $q = 0$ (i.e., only non-conjoined prefixes) for the complete level, the table size for level v is

$$S(v) = 2^{(v-L_T)}(L_T + \log_2 k). \tag{3}$$

Note that the amount of memory estimated by (2) and (3) are the size of complete prefix tables, which are implemented as memory blocks. These estimations are the provisioned amount of memory. These memory blocks are larger than the amount of memory occupied by the prefixes of a given level. We estimate the amount of memory this way as this is the method used for sizing the memory blocks needed to build an actual lookup engine. Therefore, this is a conservative estimate. Yet, we show that the amount of memory is small under real routing tables.

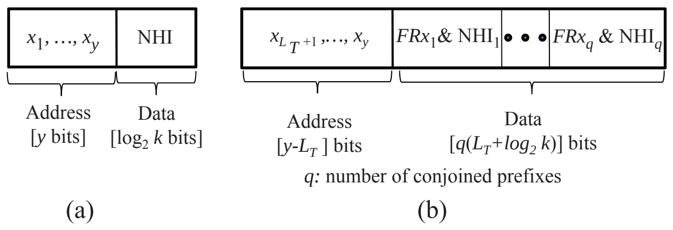


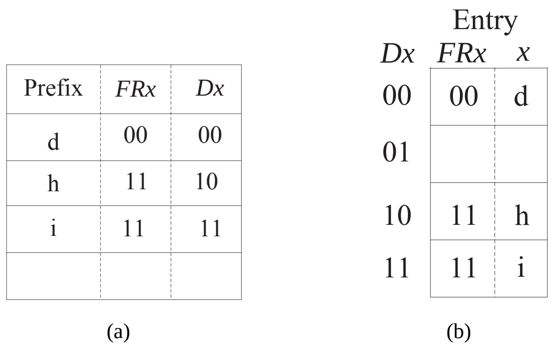
Figure 6: Format of data in prefix tables for (a) level $v \leq L_T$, and (b) level $v > L_T$ with conjoined prefixes.

3.3.2. Selection of L_T for each prefi level

A suitable L_T for a given prefix level v must satisfy two objectives: 1) To decrease the memory length (i.e., tree's height) to represent prefixes at level v. As L_T increases, the memory length decreases, as (2) and (3) show. 2) To keep the number of conjoined prefixes, or memory width, within a feasible number.

Because each prefix level is represented by a self-contained prefix table and is independent of any other prefix level, each prefix level may select a particular torsion level to achieve the two objectives stated above. We show how different prefix levels may select different L_T in another example. Let us now represent prefix level 4 with a torsion on level 2, and prefix level 6 with a torsion on level 4. Figure 7(a) shows the prefixes of level 4 with a torsion on level 2. Figure 7(b) shows prefix table for level 4 and its contents. These figures show the reduction of the memory length, from 16 locations to four. Also, the table shows that there are not-conjoined prefixes after this reduction.

Figure 8(a) shows the prefixes on level 6 with a torsion on level 4. Figure 8(b) shows the prefix table for level 6 and its contents. The table now has four locations (a tree of height two), where there are up to two conjoined prefixes. In reference to the original tree, the memory length to represent the longest prefixes in this example is reduced 16 times (from 64 to 4 locations), or exponentially reduced. On the memory width, the number conjoined prefixes didn't change (from a torsion on level 2) but the number of bits representing the family roots increased from 2 to 4 bits, but that is a small memory addition as compared to the exponential reduction of the memory length.



Prefix	*FRx*	*Dx*
d	00	00
h	11	10
i	11	11

(a)

	Entry	
Dx	*FRx*	*x*
00	00	d
01		
10	11	h
11	11	i

(b)

Figure 7: (a) Prefixes of level 4 with a torsion on level 2 and (b) prefix table of level 4.

4. Performance Evaluation

The performance of Helix is analyzed in terms of the number of memory accesses to perform: a) prefix lookup, b) prefix addition, c) prefix removal, and d) prefix modification. We also evaluate the memory amount required by Helix for different routing tables.

4.1. Lookup Speed

Helix was evaluated with three routing tables; two actual IPv4 tables and a synthetic IPv6 table. The first IPv4 table is a legacy Mae East routing table, recorded on August 24, 1997 (labeled as IPv4 Mae East) [49], holding 39,000

Prefix	*FRx*	*Dx*
f	0000	$\boxed{01}$
g	0001	10
j	1110	$\boxed{01}$
l	1111	11

(a)

	Entry 1		Entry 2	
Dx	FRx_1	x_1	FRx_2	x_2
00				
01	0000	f	1100	j
10	0001	g		
11	1111	l		

(b)

Figure 8: (a) Prefixes of level 6 with a torsion on level 4, and (b) prefix table of level 6.

IPv4 prefixes. The second table is a BGP table of AS6447 (labeled as IPv4 AS6447), recorded on May 8, 2014, holding 512,104 IPv4 prefixes [4].

The IPv6 routing table (labeled as IPv6) is generated according to a well-accepted model [50] from the IPv4 table of AS65000, also recorded on May 8, 2014 [4]. This table holds 389, 956 prefixes. We generate this synthetic IPv6 routing table because current actual IPv6 tables hold a very small number of prefixes [4].

We select torsion levels for the prefix levels under the constraint of using a memory width of up to 256 bits for the IPv4 tables to demand a conservative memory width. We select torsion levels for the IPv6 table such that the memory width is limited to 1024 bits as we target the implementation Helix on an FPGA (Section 5 includes a detailed discussion on the implementation of these three routing tables). The prefix tables for all tested routing tables are sized such that they are independently accessible; they are large enough to store all prefixes and small enough to be accessible simultaneously. Therefore, IP lookup is performed in a single memory access. The sizing of the prefix tables is discussed in detail in Section 4.3.

We also tested Helix on IPv4 and IPv6 AS6447 routing tables, with 331K and 3.1K prefixes, respectively, recorded on May 2010. The prefixes were be represented in the proposed data structure to perform lookup in a single memory access.

4.2. Table-Modificatio Speed

We consider that a prefix table may undergo the following modifications, triggered by route updates: Prefix addition, prefix removal, and prefix updates. Modifications to the prefix tables are simple because prefixes remain in their original form (original length).

Prefix addition. The addition of a new prefix requires the identification of the level of the prefix and accessing the prefix table of the corresponding level. The addition of a prefix with length y affects only the prefix table for that level; therefore, this process requires identifying the memory location and L_T, and to access the table. If $y \le L_T$, the y bits of the new prefix are used to address the location and the new NHI is added. If $y > L_T$, bits $x_{L_T+1}, \ldots, x_y$ are used to address the location, and the NHI is added together with FRx. The complexity of this operation is $O(1)$, and it is performed in one memory access.

Prefix removal. The process of removing a prefix from a prefix table is very similar to the addition operation, with the difference that the NHI and family roots are removed from the prefix table. This process also takes one memory access.

Prefix update. A prefix is updated if there is a change of NHI, prefix length, or both. The process to perform a change of NHI is similar to prefix addition (or removal). In that case, the corresponding prefix table is accessed and modified in a single memory access. If a prefix length changes, two prefix tables are modified. Because prefix tables are independent, the addition and removal are performed simultaneously. In summary, the complexity of a prefix update is also $O(1)$, and it is performed in one memory access.

4.3. Memory Usage

We also analyzed the memory usage of Helix for Mae East, IPv4 AS6447, and IPv6 routing tables. We use eight bits to represent up to 256 ports, or NHIs, which are enough for a large router. The memory amount is estimated as the amount used by the total provisioned memory. The memory amount is given in bytes.

Memory Used for Mae East 1997. Figure 9(a) shows the prefix distribution of the routing table on the different prefix lengths. Figures 9(b), 9(c), and 9(d) show the selected torsion levels for each prefix length, the number of conjoined prefixes per level, and the provisioned memory per level, respectively. A torsion level for each level is selected to allow the largest number of conjoined prefixes that falls within the memory width of the selected constraint (i.e., 256 bits). The total amount of memory provisioned for the Mae-East 1997 is 944 Kbytes.

Memory Used for IPv4 AS6447 2014. Figure 10(a) shows the prefix distribution of the routing table. The figure also shows that prefixes are found on levels 8 to 32. From those, levels 16 to 24 have the larger number of prefixes where level 24 is the most populated. A torsion level for each prefix level is selected (Figure 10(b)) within the width constrain. Figure 10(c) shows the number of conjoined prefixes per level. Figure 10(d) shows the memory amount provisioned for each level. This figure also shows that level 24 requires 1.4 Mbytes, which is the largest amount of memory among all levels. The total amount of memory Helix requires for this routing table is 3.4 Mbytes.

Memory Used for IPv6 forwarding table.

The studied IPv6 table holds prefixes with lengths from 17 to 64 bits, as Figure 11(a) shows. The figure also shows that level 48 holds the largest number of prefixes; 113, 750. The selection of the torsion levels for this table is performed by keeping the largest number of conjoined prefixes such that the largest memory width is equal to or smaller than 1024 bits. We select this width as we target the implementation on an FPGA (discussed in detail in Section 5). The FPGA permits the aggregation of several RAM blocks to build a very wide memory. Figure 11(b) shows the selected torsion levels per prefix level. The torsion level for level 48 is 34. Figure 11(c) shows the largest number of conjoined prefixes for each prefix level. Level 48 has 19 conjoined prefixes, as the figure shows. Figure 11(d) shows the total amount of memory per level. Prefix level 48, being the most populated, is provisioned with a memory block of about 1.6 Mbytes of memory. The total amount of required memory for this table is 5.2 Mbytes.

5. Implementation of Helix on an FPGA

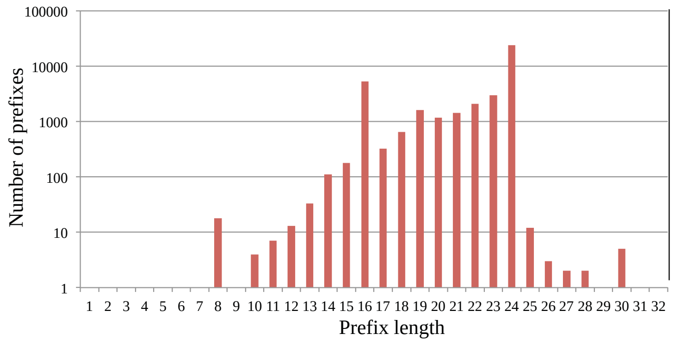
Allocating a memory block for each prefix level and using small memories per level are two attractive properties for implementing Helix on a single FPGA. These programmable devices flexibly avail a large number of blocks of random-access memory (RAM). For example, the Xilinx Virtex-7 XC7VX1140T FPGA provides 2,128 RAMB18E1/FIFO18E1 or 1,064 RAMB36E1 memory blocks [51]. The available on-chip memory allows us to avoid using external memory.

We implemented the two studied IPv4 tables and the IPv6 table on the FPGA. The memories per levels required by Helix were mapped onto the FPGA RAM blocks to evaluate both the number of memory blocks used, the memory utilization (i.e., the ratio of the number of RAM blocks used and the total number of RAM blocks available in the FPGA), and the utilization of FPGA Lookup tables (LUTs).

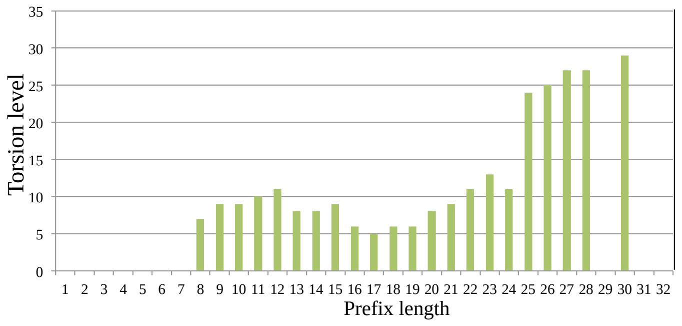
Table 2 shows the number of RAM blocks allocated, post-synthesis, to each routing table and the memory utilization for each of the routing tables. This table shows the routing tables comfortably fit in the on-chip memory of a single FPGA. As the memory utilization indicates, the IPv6 routing table demands the largest amount resources, or 72% of the available on-chip memory.

Different from the amount of consumed memory, the number of LUTs of the FPGA is larger for the IPv4 AS6447 table that for the IPv6 table. This is expected because the average difference between the selected torsion levels and the corresponding prefix level is larger in the IPv4 tables. This is a result of the distribution of prefixes. In fact, the results show that the distribution of IPv6 prefixes is more benign (for building a Helix structure) than the distribution of prefixes in the IPv4 tables because of the larger address space.

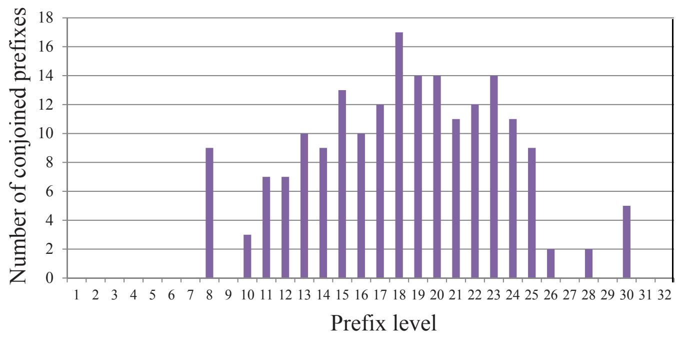
The post place-and-route time analysis performed through Xilinx synthesis tool shows a working frequency of 632 MHz for the IPv4 routing tables and 506 MHz for the IPv6 routing table. These clock speeds can be represented



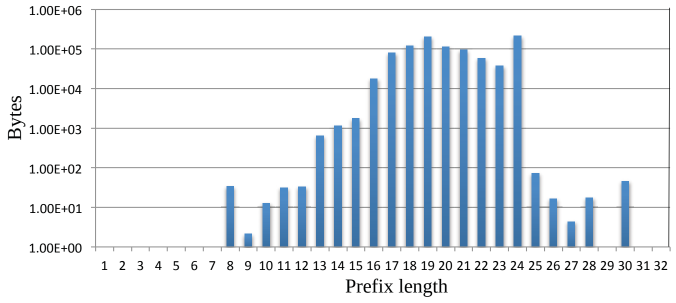
(a) Prefix distribution of IPv4 Mae East



(b) Torsion levels per prefix length

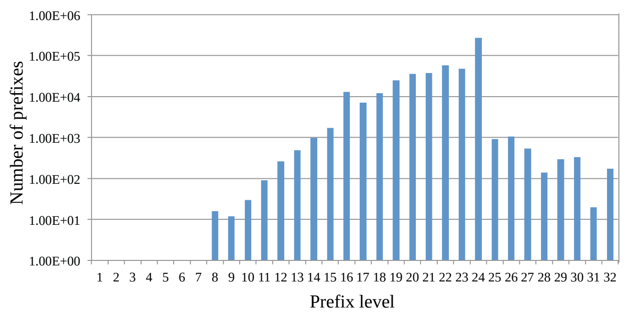


(c) Number of conjoined prefixes

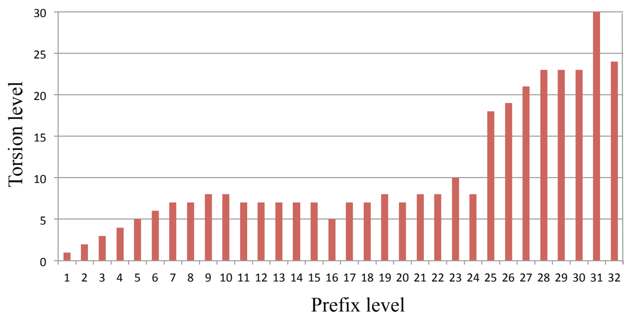


(d) Memory required per level

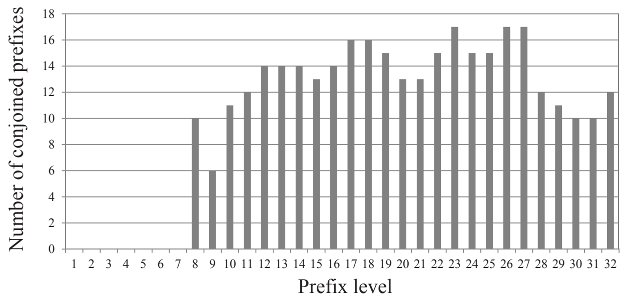
Figure 9: Helix of IPv4 Mae East with 39K prefixes.



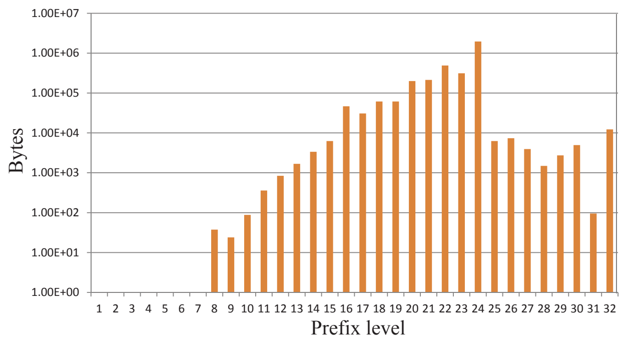
(a) Prefix distribution of IPv4 AS6447



(b) Torsion levels per prefix length

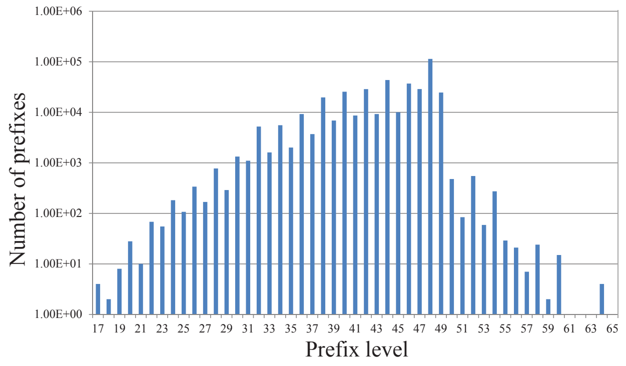


(c) Number of conjoined prefixes

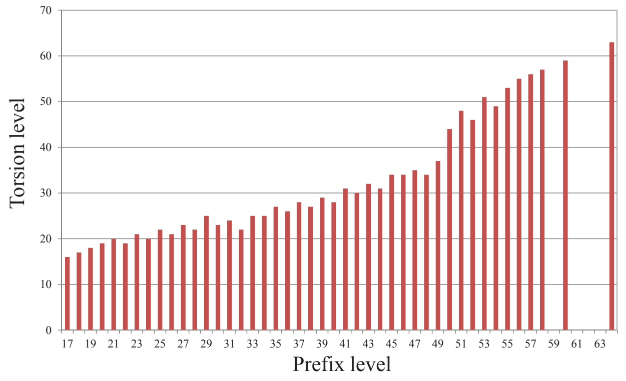


(d) Memory required per level

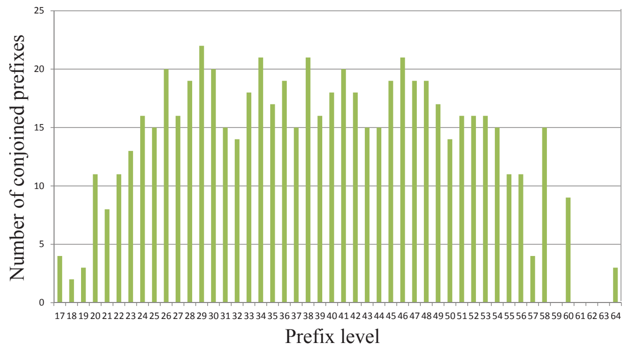
Figure 10: Helix of IPv4 AS6447 2014 table with 512K prefixes.



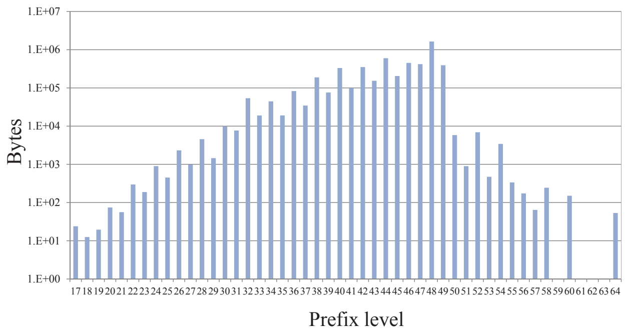
(a) IPv6 prefix distribution



(b) Smallest torsion level per prefix length



(c) Number of conjoined prefixes



(d) Memory required per level

Figure 11: Helix of IPv6 table with 389K prefixes.

Table 2: Memory provisioning in Helix for different routing tables.

Table	Estimated memory (bytes)	No. 36-Kbit blocks (utilization)	No. LUTs (utilization)
IPv4 Mae East	944K	244 (12%)	2,563 (<1%)
IPv4 AS6447	3.4M	853 (45%)	17,873 (2%)
IPv6	5.2M	1,354 (72%)	10,280 (3%)

in packets per second (pps). Therefore, Helix may process beyond 632 Mpps for the IPv4 routing tables and 506 Mpps for the IPv6 routing table, all using a single lookup engine and 50-byte packets. Moreover, the FPGA has room to accommodate up to eight engines for the Mae East routing table and two engines for the 2014 IPv4 routing table, multiplying the achievable lookup speeds by the same factors, or a lookup speed larger than 1.2 Gpps for both routing tables. The number of engines are limited only by the amount of memory available in the FPGAs as the amount of LUTs is dramatically small. In any case, the single memory access used by Helix allows us to reach much larger speeds than the number of IP lookups required by optical links running at 100 Gbps (250 Mpps on 50-byte packets).

5.1. Implementation Notes

As described in Section 5, the implementation and design of Helix was tested on an FPGA design. Helix has low complexity as prefixes are not encoded and there is a memory block dedicated to each prefix level. Therefore, one of the important blocks in the design of Helix is the comparators and detector of the longest matching prefix. Figure 12 shows the design of a match detector for a prefix level. This detector is used to indicate if a match is found in a prefix level (v). The destination of the packet arriving in the router is held in a register and bits $v - L_T$ are used to address the prefix table (RAM block). The output of the prefix table may have up to q entries and their contents are compared to the selected bits of the destination address. The contents are the matching address (MA), output port (OP), and a valid bit (VB) for each of the conjoined q prefixes. The Memory Address (MA) block selects the bits needed for addressing prefix table v and the Match bits block selects the family root bits of the packet's destination address. The comparison results are aggregated through the OR gate and the output indicates whether there is a match. The simplicity of the scheme allows to perform gate-level design, ensuring a short response time.

6. Comparison of Memory used by Helix and other Schemes

To find out whether the improved lookup speed of Helix have an associated cost on memory, we evaluated the memory usage of memory-efficient schemes, such as: LC-trie, Multiway (6-way), and a recent schemes; Offset Encoded Trie (OET) [47] all of which require multiple memory accesses for performing IP lookup and route updates under the considered routing tables.

For the legacy IPv4 Mae East routing table, the amounts of memory required are 1.2 Mbyte by LC-trie, 2.3 Mbyte by Multiway, 200 Kbyte by OET, and 944 Kbyte by Helix. For the 2014 IPv4 AS6447 routing table, the required amounts of memory are 11.6 Mbyte by LC-trie, 5.14 Mbyte by Multiway, 2.4 Mbyte by OET, and 3.4 Mbyte by Helix.

For the synthetic IPv6 routing table, the amounts of memory required are 13.9 Mbyte by LC-trie, 4.17 Mbyte by Multiway, and 5.2 Mbyte by Helix. Our system was not able to estimate the memory required by the OET scheme as this scheme is not scalable to long prefixes and large IPv6 routing tables, as those studied in this paper, as the generation of encoded bit maps requires very complex processing.

The results show that in comparison with legacy and recent schemes, Helix requires very small amounts of memory. In addition, Helix also accommodates table updates in $O(1)$ complexity while some memory-efficient schemes require long processes to accommodate these updates. In this way, Helix is also scalable for long prefixes and large routing tables.

Other recent schemes, which have the objective to reduce the amount of required memory, require multiple memory accesses for IP lookup or table updates [52, 53, 54, 55, 56, 57, 58]. However, lookup speed remains the major feature required by lookup engines for the increasingly higher data rates in the Internet.

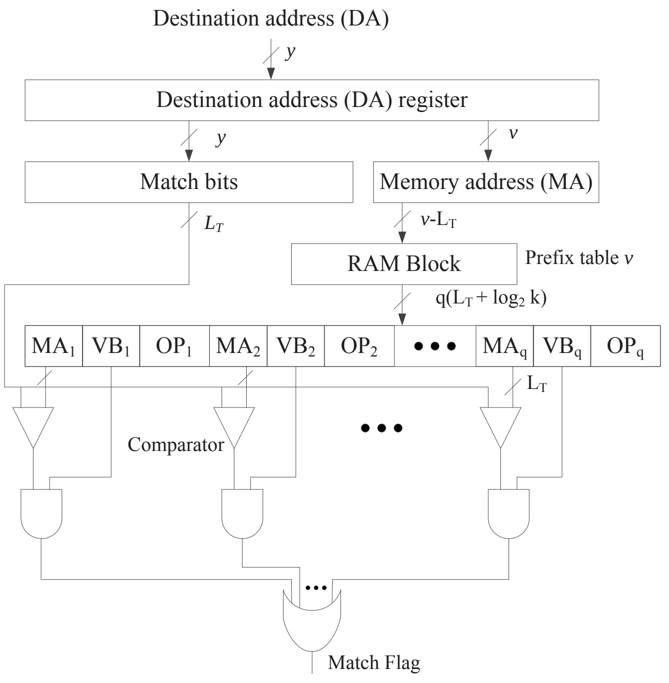


Figure 12: Comparator design.

7. Conclusions

In this paper, we proposed an IP lookup scheme, called Helix, to represent prefixes of a forwarding table in small memories. Helix searches for matching prefixes at all prefix levels in parallel and uses helicoidal properties of the binary trees to succinctly represent prefixes. The result is an IP lookup scheme that achieves longest prefix matching in a single memory access while using a small amount of memory. In our evaluation with actual routing tables, we showed that Helix performs IP lookup for IPv4 and IPv6 prefixes in one memory access. These tables are a legacy Mae East routing table (1997), a recent IPv4 routing table with 512K prefixes (2014), and a synthetic IPv6 routing table with 389K prefixes.

Helix is simple as it doesn't require to code nor aggregate prefixes, and yet, it is efficient in both the number of memory accesses and amount of memory used. These properties also permit updates, triggered by changes in a routing table, to the prefix tables in a single memory access.

We implemented Helix on an FPGA and the amount of resources (memory and logic blocks) of the FPGA required is up to 72% for an IPv6 routing table with 389K prefixes, while achieving a very high clock frequency. These results show that large routing tables can be implemented in a single FPGA, without resorting to external memory and while keeping the single memory-access time for lookup and table updates.

The introduced helicoidal properties are used conservatively in this paper to keep the description simple. The benefits of using Helix on decreasing the required amount of memory and bounding the number of memory accesses to one are a significant improvement to IP lookup. These properties might be further explored to achieve even greater memory savings.

[1] F. Baker, Requirements for IP version 4 routers, RFC 1812, June 1995.
[2] P. Gupta, S. Lin, N. McKeown, Routing lookups in hardware at memory access speeds, Proc. IEEE INFOCOM (1998) 1240–1247.
[3] J. McDonough, Moving standards to 100 gbe and beyond, Communications Magazine, IEEE 45 (11) (2007) 6–9.
[4] Routing tables, `http://bgp.potaroo.net`; accessed May, 2014. (June 2014).
[5] M. Zitterbart, T. Harbaum, D. Meier, D. Brokelmann, Efficient routing table lookup for IPv6, in: High-Performance Communication Systems, 1997.(HPCS'97) The Fourth IEEE Workshop on, IEEE, 1997, pp. 1–9.
[6] P. Warkhede, S. Suri, G. Varghese, Multiway range trees: scalable IP lookup with fast updates, Computer Networks 44 (3) (2004) 289–303.

[7] H. Song, J. Turner, J. Lockwood, Shape shifting tries for faster IP route lookup, in: Network Protocols, 2005. ICNP 2005. 13th IEEE International Conference on, 2005, pp. 10 pp.–367. doi:10.1109/ICNP.2005.36.
[8] H. Song, F. Hao, M. Kodialam, T. V. Lakshman, IPv6 lookups using distributed and load balanced bloom filters for 100gbps core router line cards, in: INFOCOM 2009, IEEE, 2009, pp. 2518–2526. doi:10.1109/INFCOM.2009.5062180.
[9] Y. Qu, V. K. Prasanna, High-performance pipelined architecture for tree-based IP lookup engine on fpga, in: Parallel and Distributed Processing Symposium Workshops PhD Forum (IPDPSW), 2013 IEEE 27th International, 2013, pp. 114–123. doi:10.1109/IPDPSW.2013.168.
[10] M. Bando, H. Chao, Flashtrie: Hash-based prefix-compressed trie for IP route lookup beyond 100 Gbps, Proc. IEEE INFOCOM (2010) 1–9.
[11] C. Labovitz, G. R. Malan, F. Jahanian, Internet routing instability, ACM SIGCOMM (1997) 115–126.
[12] D. Yu, B. Smith, B. Wei, Forwarding engine for fast routing lookups and updates, Proc. IEEE GLOBECOM (1999) 1556–1563.
[13] D. Taylor, J. Lockwood, T. Sproull, J. Turner, D. Parlour, Scalable IP lookup for programmable routers, Proc. IEEE INFOCOM 21 (1) (2002) 562–571.
[14] G. Huston, T. Bates, P. Smith, CIDR report, Web site, http://www. cidr-report. org.
[15] R. Sangireddy, N. Futamura, S. Aluru, A. K. Somani, Scalable, memory efficient, high-speed IP lookup algorithms, IEEE/ACM Trans. on Networking 13 (4) (2005) 802–812.
[16] H. Lim, C. Yim, E. E. Swartzlander, Priority tries for IP address lookup, IEEE Trans. on Computers 59 (6) (2010) 784–794.
[17] M. Degermark, A. Broadnik, S. Carlsson, S. Pink, IP-address lookup using lc-tries, ACM SIGCOMM (1997) 3–14.
[18] M. Waldvogel, G. Varghese, J. Turner, B. Plattner, Scalable high speed IP routing lookups, ACM SIGCOMM (1997) 25–36.
[19] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Trans. on Networking 7 (3) (1999) 324–334.
[20] N.-F. Huang, S.-M. Zhao, A novel IP-routing lookup scheme and hardware architecture for multigigabit switching routers, IEEE J. Select. Areas Commun. 17 (6) (1999) 1093–1104.
[21] P. Gupta, B. Prabhakar, S. Boyd, Near-optimal routing lookups with bounded worst case performance, Proc. IEEE INFOCOM (2000) 1180–1192.
[22] S. Nilsson, G. Karlsson, IP-address lookup using LC-tries, IEEE J. Select. Areas Commun. 17 (6) (1999) 1083–1092.
[23] R. Sangireddy, N. Futamura, S. Aluru, A. K. Somani, Scalable, memory efficient, high-speed IP lookup algorithms, IEEE/ACM Trans. on Networking 13 (4) (2005) 802–812.
[24] I. Ioannidis, A. Grama, M. Atallah, Adaptive data structures for IP lookups, Proc. IEEE INFOCOM (2003) 75–84.
[25] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, in: Proceedings of the ACM SIGCOMM '97 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communication, SIGCOMM '97, ACM, New York, NY, USA, 1997, pp. 3–14. doi:10.1145/263105.263133.
URL `http://doi.acm.org/10.1145/263105.263133`
[26] R. Rojas-Cessa, L. Ramesh, Z. Dong, L. Cai, N. Ansari, Parallel-search trie-based scheme for fast IP lookup, Proc. IEEE GLOBECOM (2007) 210–214.
[27] J. Zhao, X. Zhang, X. Wang, Y. Deng, X. Fu, Exploiting graphics processors for high-performance IP lookup in software routers, in: INFOCOM, 2011 Proceedings IEEE, 2011, pp. 301–305. doi:10.1109/INFCOM.2011.5935144.
[28] V. Srinivasan, G. Varghese, A survey of recent IP lookup schemes, in: Protocols for High-Speed Networks VI, Springer, 2000, pp. 9–23.
[29] H. J. Chao, Next generation routers, Proceedings of the IEEE 90 (9) (2002) 1518–1558.
[30] G. Varghese, Network algorithmics, Chapman & Hall/CRC, 2010.
[31] F. Zane, G. Narlikar, A. Basu, Coolcams: power-efficient TCAMs for forwarding engines, in: INFOCOM 2003. Twenty-Second Annual Joint Conference of the IEEE Computer and Communications. IEEE Societies, Vol. 1, 2003, pp. 42 – 52 vol.1. doi:10.1109/INFCOM.2003.1208657.
[32] D. R. Morrison, Patricia: practical algorithm to retrieve information coded in alphanumeric, Journal of the ACM (JACM) 15 (4) (1968) 514–534.
[33] S. Nilsson, G. Karlsson, Fast address look-up for internet routers., in: Broadband Communications, Citeseer, 1998, pp. 11–22.
[34] B. Lampson, V. Srinivasan, G. Varghese, IP lookups using multiway and multicolumn search, IEEE/ACM Trans. on Networking 7 (3) (1999) 324–334.
[35] W. N. Eatherton, Hardware-based internet protocol prefix lookups, master's thesis, sever institute, washington university, st, Louis, MO, May.
[36] M. Degermark, A. Brodnik, S. Carlsson, S. Pink, Small forwarding tables for fast routing lookups, SIGCOMM Comput. Commun. Rev. 27 (4) (1997) 3–14. doi:10.1145/263109.263133.
URL `http://doi.acm.org/10.1145/263109.263133`
[37] N.-F. Huang, S.-M. Zhao, J.-Y. Pan, C.-A. Su, A fast IP routing lookup scheme for gigabit switching routers, in: INFOCOM'99. Eighteenth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, Vol. 3, IEEE, 1999, pp. 1429–1436.
[38] A. Broder, M. Mitzenmacher, Using multiple hash functions to improve IP lookups, in: INFOCOM 2001. Twentieth Annual Joint Conference of the IEEE Computer and Communications Societies. Proceedings. IEEE, Vol. 3, IEEE, 2001, pp. 1454–1463.
[39] J. Van Lunteren, Searching very large routing tables in wide embedded memory, in: Global Telecommunications Conference, 2001. GLOBECOM'01. IEEE, Vol. 3, IEEE, 2001, pp. 1615–1619.
[40] S. Dharmapurikar, P. Krishnamurthy, D. E. Taylor, Longest prefix matching using bloom filters, in: Proceedings of the 2003 conference on Applications, technologies, architectures, and protocols for computer communications, ACM, 2003, pp. 201–212.
[41] H. Song, S. Dharmapurikar, J. Turner, J. Lockwood, Fast hash table lookup using extended bloom filter: An aid to network processing, in: Proceedings of the 2005 Conference on Applications, Technologies, Architectures, and Protocols for Computer Communications, SIGCOMM '05, ACM, New York, NY, USA, 2005, pp. 181–192. doi:10.1145/1080091.1080114.
URL `http://doi.acm.org/10.1145/1080091.1080114`
[42] A. Alagu Priya, H. Lim, Hierarchical packet classification using a bloom filter and rule-priority tries, Computer Communications 33 (10) (2010) 1215–1226.
[43] H. Lim, K. Lim, N. Lee, K. Park, On adding bloom filters to longest prefix matching algorithms, Computers, IEEE Transactions on PP (99)

(2012) 1. doi:10.1109/TC.2012.193.
[44] H. Le, V. Prasanna, Scalable tree-based architectures for IPv4/v6 lookup using prefix partitioning, Computers, IEEE Transactions on 61 (7) (2012) 1026–1039. doi:10.1109/TC.2011.130.
[45] H. Song, M. Kodialam, F. Hao, T. Lakshman, Scalable IP lookups using shape graphs, in: Network Protocols, 2009. ICNP 2009. 17th IEEE International Conference on, IEEE, 2009, pp. 73–82.
[46] V. Srinivasan, G. Varghese, Faster IP lookups using controlled prefix expansion, ACM SIGMETRICS Performance Evaluation Review 26 (1) (1998) 1–10.
[47] K. Huang, G. Xie, Y. Li, A. X. Liu, Offset addressing approach to memory-efficient IP address lookup, in: INFOCOM, 2011 Proceedings IEEE, IEEE, 2011, pp. 306–310.
[48] S. Dandamudi, Fundamentals of computer organization and design, Springer, 2003.
[49] Mae East routing table (1997) [online], `http://www.nada.kth.se/~snilsson/software/router/Data/.`; accessed June, 2010. (1997).
[50] M. Wang, S. Deering, T. Hain, L. Dunn, Non-random generator for IPv6 tables, in: High Performance Interconnects, 2004. Proceedings. 12th Annual IEEE Symposium on, IEEE, 2004, pp. 35–40.
[51] Xilinx Inc., Virtex-7 data sheet, `http://www.xilinx.com/products/silicon-devices/fpga/virtex-7.html`.
[52] H. Lim, C. Yim, E. Swartzlander, Priority tries for IP address lookup, Computers, IEEE Transactions on 59 (6) (2010) 784–794. doi:10.1109/TC.2010.38.
[53] Z. Čiča, A. Smiljanić, Balanced parallelised frugal IPv6 lookup algorithm, Electronics letters 47 (17) (2011) 963–965.
[54] S.-Y. Hsieh, Y.-C. Yang, A classified multisuffix trie for IP lookup and update, Computers, IEEE Transactions on 61 (5) (2012) 726–731. doi:10.1109/TC.2011.86.
[55] D. Pao, Z. Lu, Y. H. Poon, IP address lookup using bit-shuffled trie, Computer Communications 47 (2014) 51–64.
[56] O. Erdem, A. Carus, H. Le, Large-scale sram-based IP lookup architectures using compact trie search structures, Computers & Electrical Engineering 40 (4) (2014) 1186–1198.
[57] Y. Li, D. Zhang, K. Huang, D. He, W. Long, A memory-efficient parallel routing lookup model with fast updates, Computer Communications 38 (2014) 60–71.
[58] K. Huang, G. Xie, Y. Li, D. Zhang, Memory-efficient IP lookup using trie merging for scalable virtual routers, Journal of Network and Computer Applications 51 (2015) 47–58.